Parallel and Distributed Computing

Joint International Masters

Prof. Dr. Ronald Moore

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Last Modified: 4/1/12

Outline of Chapter I – Introduction

Changes in the course of the semester are possible (probable, even).

Personal Introductions – Instructor (1)

Instructor: Prof. Dr. Ronald Charles Moore

1983	B.Sc. Michigan State Univ.
1983-89	Software Engineer Texas Instruments: Design Automation Div.
1989	Came to Germany
1995	*Diplom-Informatiker* Goethe-Univ. (Frankfurt)
2001	PhD Goethe-Univ. (Frankfurt)
2001-7	Product Mgr (among other things) with an Internet Company in Frankfurt

since Sept. 2007 with the FBI
(a.k.a. *Fachbereich Informatik*)

Research: Compilers & Architectures for exploiting implicit parallelism

Business: Web-based Software as a Service (SaS), serving Financial Market Data Apps...

Personal Introductions – Instructor (2)

Instructor: Prof. Dr. Ronald Charles Moore

Official Subject Areas (*Fachgebiete*):

- Foundations of Informatics
- Net-Centric Computing

That means Operating Systems, Distributed Systems among other things

I'm still trying to decide what that means – but this course belongs to it!

Personal Introductions – Instructor (3)

Instructor: Prof. Dr. Ronald Charles Moore

Yes, English is my mother-tongue...

...but I do speak German and I'll take questions in either English or German

...but we have participants who do *not* speak German, so let's try to stick to English most of the time.

...it is *your responsibility to tell me* should I:
- → speak too quickly,
- → use colloquialisms
- → ...or terminology you're not familiar with,
- → or just generally stop making sense!

Such as "colloquialisms" perhaps?

Personal Introductions – Students

Your Background:

0. What have you learned so far (e.g. in your Bachelors)?
1. Have you had a chance to program parallel and/or distributed systems already?
2. What do you expect from this course?
3. What would you *like* to learn in this course?

Outline of Chapter I – Introduction

I Introduction

II Models of Parallel Computing

III Parallel Computation Design

IV The Message-Passing Paradigm

V The Shared Memory Paradigm

VI Frontiers

VII Summary

A) Personal Introductions

B) What is Parallel & Distributed Computing?

C) Demotivation: Why wasn't Parallelism used?

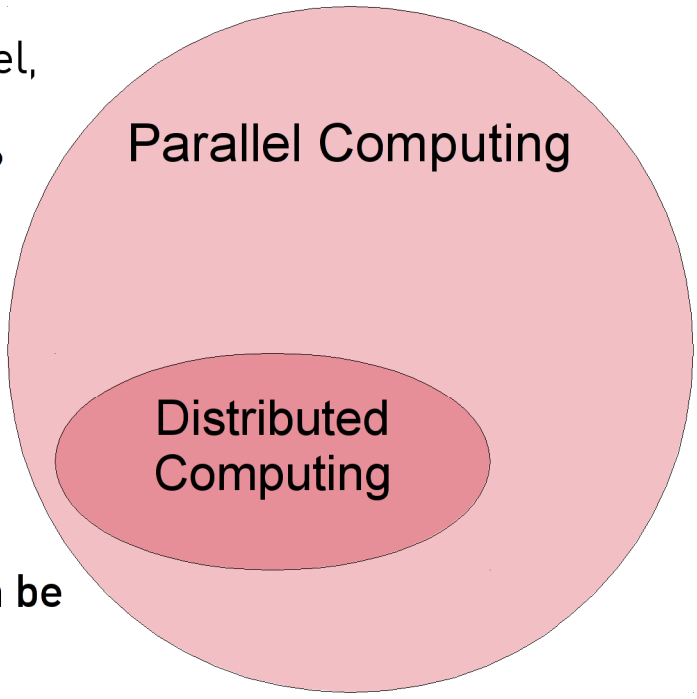
D) Motivation: Why use Parallelism?

E) Paradigms & Packages for Parallel Programming: An Overview & an Outlook

F) Summary & Bibliography

What is Parallel & Distributed Computing?

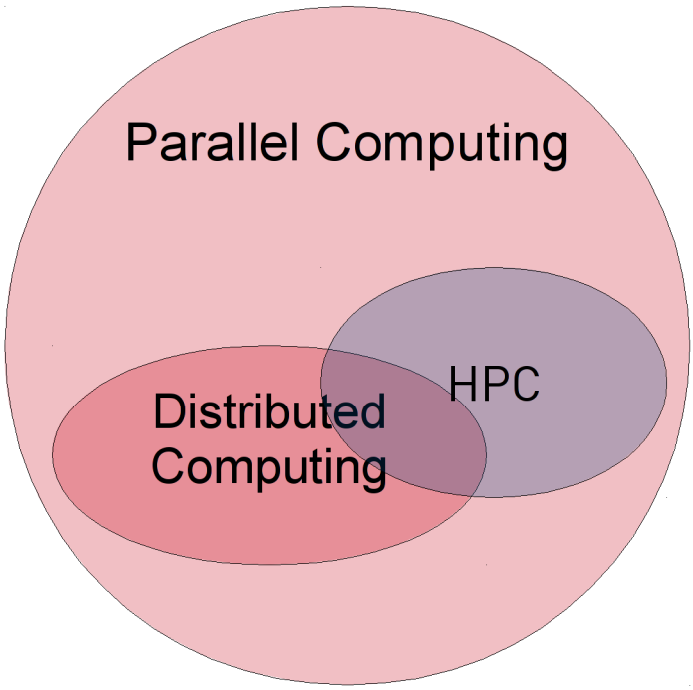
- A Term you always see like that (seeming redundant!).
- Distributed Computing is Parallel, and Parallel Computing is Distributed, by definition! Right?
- We will take Distributed Computing to be computing which is distributed to serve a purpose (other than being parallel) – normally overcoming geographical seperation.
- **Thus Distributed Computing can be taken to be a subset of Parallel Computing**



What *else* is Parallel Computing?

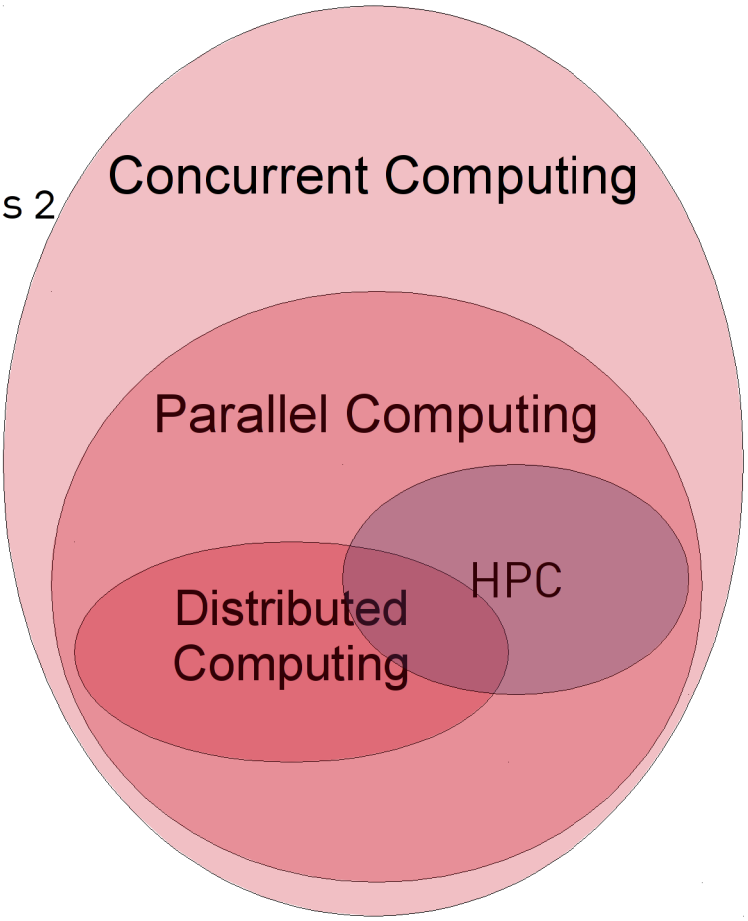
- **Why a subset?** Why else build or use parallel computers?
- One word: **Performance!**
- The other subset of Parallel Computing is **High Performance Computing (HPC).**

The focus of this course is on Parallelism, both in the sense of Distributed Computing and in the sense of HPC.



Parallel vs. Concurrent

- But I haven't defined parallel computing (did you notice?)
- Computing is **parallel** as soon as 2 (or more) operations can take place at the same time. This usually implies multiple CPUs (Cores).
- Computing is **concurrent** as soon as 2 (or more) operations can take place in any order, i.e. they are not placed in a strict **sequential** order. This implies multiple **threads** and/or multiple **processes.**

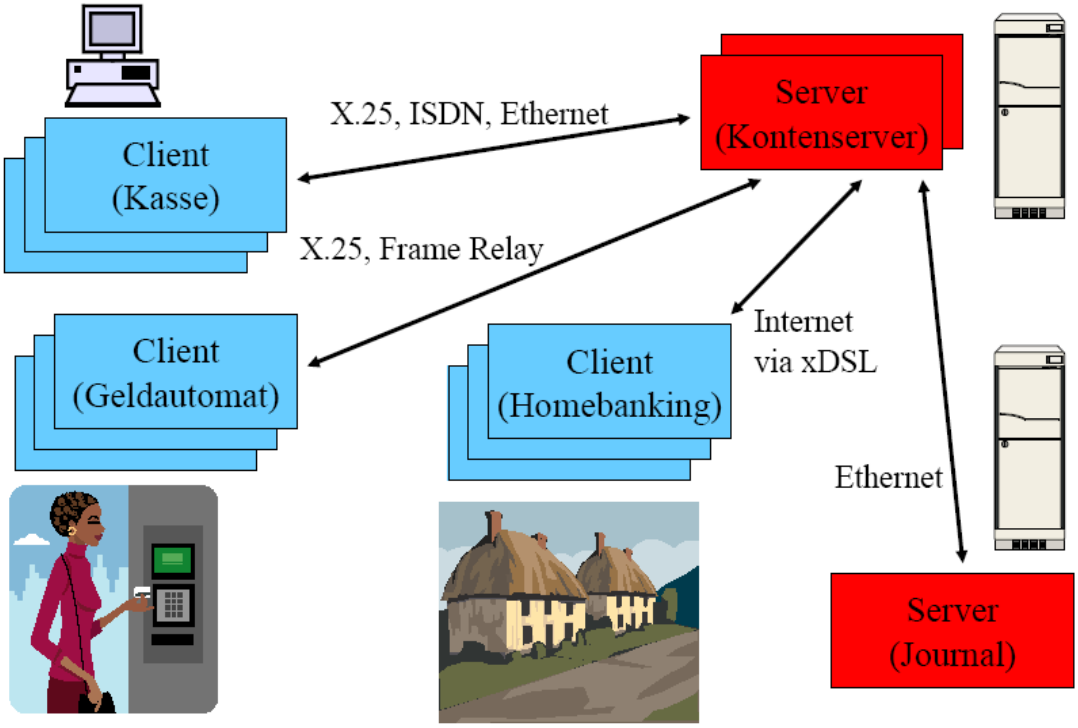


Example (1) - Distributed!

The private network connecting the various branches of a bank to the central office.

Meanwhile, extended to accommodate shops, home banking, etc.

(Pardon the German).



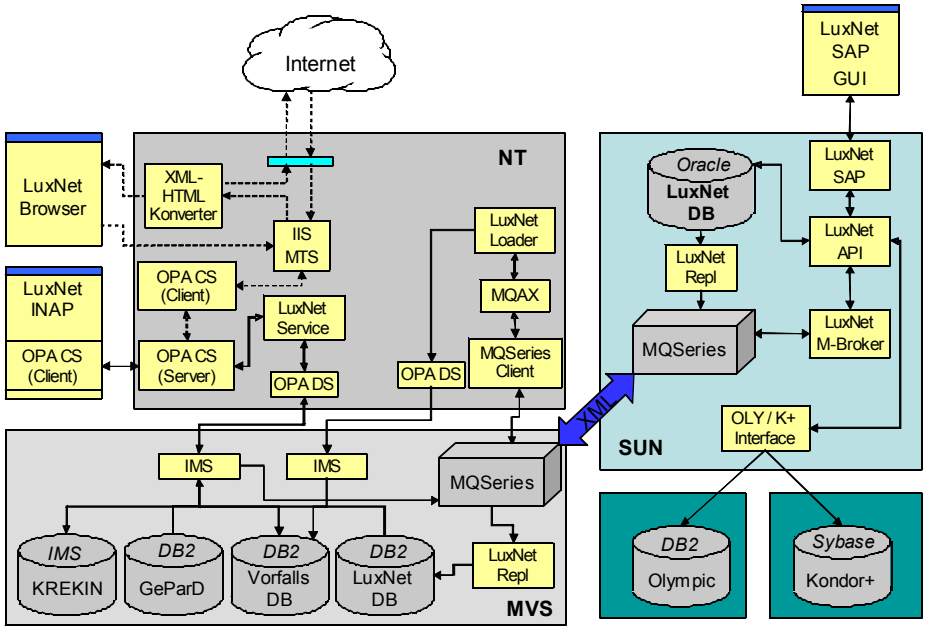
Source: “Verteilte Systeme”, Skript, Peter Wollenweber, Hochschule Darmstadt, Fachbereich Informatik.

Examples (2) - Distributed?

The computer center of a large bank – full of servers with connections to the branch offices.

Very heterogeneous.

Much of the architecture consists of *legacy systems.*

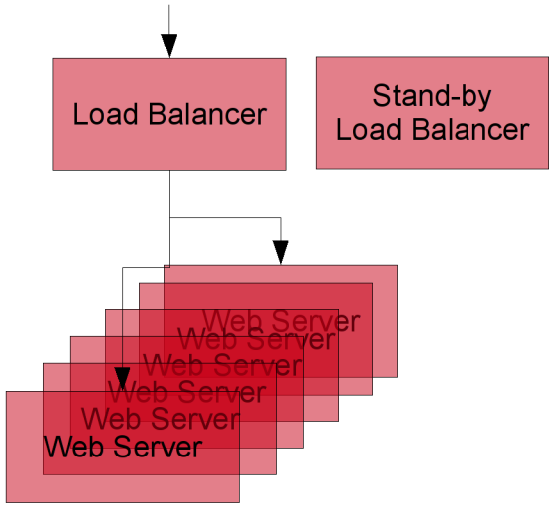


Source: “Verteilte Systeme”, Skript, Alois Schütte, Hochschule Darmstadt, Fachbereich Informatik.

Examples (3) – A Web Server Farm

- **The Internet?** Many systems or one big system?

  A "Production" Web-Site is often hosted by a "web server farm" - a.k.a. a "load-balanced cluster" consisting of:
  - 2 (or more) Load Balancers
  - Many Web Servers
  - Database Servers
  - Possibly connections to other servers, services...



Examples (4) –The Googleplex

Google: supports several computer centers with more than 200,000 PC's (as of 2003, meanwhile a lot more!).

Servers:
- Load Balancers
- Proxy Servers (caches)
- Web Servers
- Index servers
- Document servers
- Data-gathering servers
- Ad servers
- Spelling servers

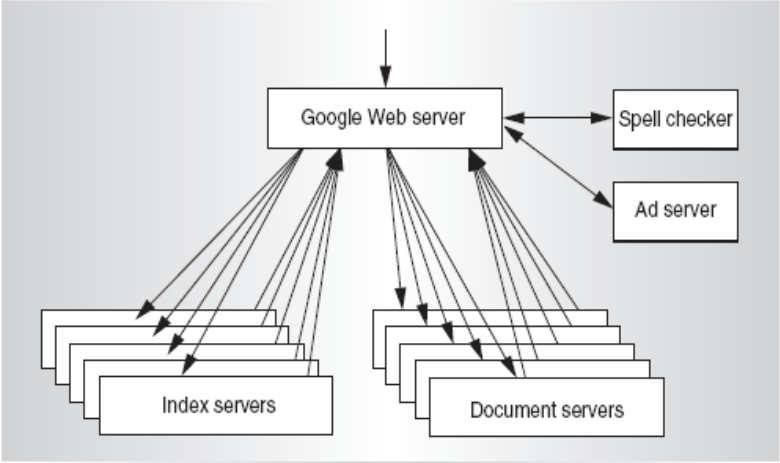


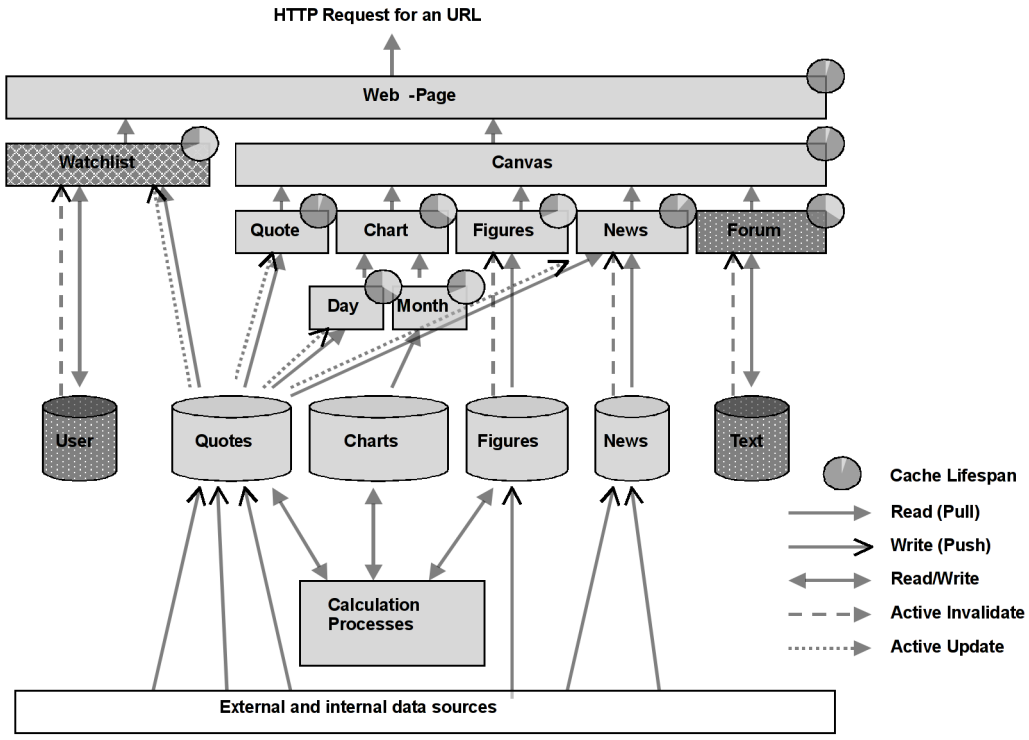
Figure 1. Google query-serving architecture.

Source: http://en.wikipedia.org/w/index.php?title=Google_platform&oldid=202504102,, Barroso et. al. W*eb Search for a Planet: The Google Cluster Architecture,* **IEEE Micro,** March-April 2003 (http://labs.google.com/papers/googlecluster-ieee.pdf)

Examples (5) – An Internet Company in Frankfurt

Financial Market Data Servers import data from data suppliers, add value, mix in data supplied by users, plus charts and quantitative analysis, and make it available as a internet or intranet application.

Note dataflow, granularity...



Source: Cotoaga, K.; Müller, A.; Müller, R.: *Effiziente Distribution dynamischer Inhalte im Web*. In **Wirtschaftsinformatik** 44 (2002) 3, p. 249-259.

Examples (6) – Deep Blue

"Deep Blue was a chess-playing computer developed by IBM. On May 11, 1997, the machine won a six-game match... against world champion Garry Kasparov."

"...It was a 32-node IBM RS/6000 SP high-performance computer, which utilized the Power Two Super Chip processors (P2SC). Each node of the SP employed 8 dedicated VLSI chess processors, for a total of 256 processors working in tandem. ... It was capable of evaluating 200 million positions per second... . In June 1997, Deep Blue was the 259th most powerful supercomputer according to the TOP500 list, achieving 11.38 GFLOPS on the High-Performance LINPACK benchmark."

Sources: "Deep Blue (chess computer)." Wikipedia, The Free Encyclopedia.15 Oct 2009 <http://en.wikipedia.org/w/index.php?title=Deep_Blue_(chess_computer)&oldid=318710273>, corrected with information from <http://www.research.ibm.com/deepblue/meet/html/d.3.shtml>.

Examples (7) – Blue Gene

"Blue Gene is a computer architecture project designed to produce several supercomputers, designed to reach operating speeds in the PFLOPS (petaFLOPS) range, and currently reaching sustained speeds of nearly 500 TFLOPS (teraFLOPS). ...

The project was awarded the National Medal of Technology and Innovation by US President Obama on September 18, 2009."

"On November 12, 2007, the first [Blue Gene/P] system, JUGENE, with 65536 processors is running in the Jülich Research Centre in Germany with a performance of 167 TFLOPS.[15] It is the fastest supercomputer in Europe and the sixth fastest in the world "

Source: "Blue Gene." Wikipedia, The Free Encyclopedia. 15 Oct 2009 <http://en.wikipedia.org/w/index.php?title=Blue_Gene&oldid=319969920>. .

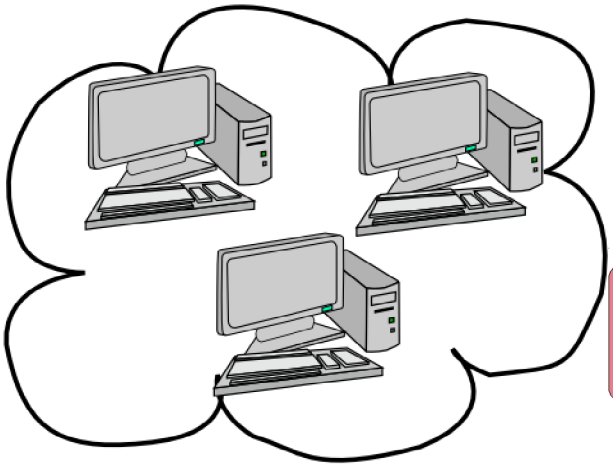
Examples (8) – HHLR

The Hessische HochLeistungs Rechner. Right here in Darmstadt.

"The system consists of 15 SMP-Nodes with a total of 452 Processors. ... The 14 computing nodes contain 32 Power6-CPUs each and at least 128 GB RAM. When used as Shared Memory computers (SMP), the nodes are suitable for applications with high communication requirements. In order to also run programs that require more than one SMP-Node, the computers are connected by a very fast internal network (8 Lanes DDR Infiniband)."

Source: " Der Hessische Hochleistungsrechner" <http://www.hhlr.tu-darmstadt.de/organisatorisches/startseite/index.de.jsp>, translation by R. C. Moore.

Examples (9) – BOINC + Grids + Clouds = ???



See **c't** (Heise Verlag), Ausgabe 21 (29.9.) 2008 Seite 128-145.

BOINC = Berkeley Open Infrastructure for Network Computing = Open infrastructure for applications such as SETI@home – exploit CPU cycles that would otherwise be wasted.

Grids = More or less open infrastructures for collaborations and resource (and data) sharing among distributed teams. Goal: Utility Computing (Role-Model: Electric Grid). Mostly limited (still) to Research Centers (e.g. CERN).

Clouds = Commecial Infrastructures (open to paying customers). Newest form of Utility Computing. Built on availability of Virtual Machines (and waste CPU cycles at Amazon, IBM, etc.).

Outline of Chapter I – Introduction

(De)Motivation: Why (not) Use Parallelism?

Why was Parallel Computing neglected for so long?

Parallelism has been omnipresent in *hardware* development (computer architecture) for decades.

But it has remained a specialty (often treated as an oddity) in *software architecture* and *software engineering* until recently (consider it's place in your education so far).

Why?

Four Factors:

1) **Moore's Law**
2) **Amdahl's Law**
3) **Inherent Difficulty**
4) **Lack of a Unifying Paradigm**

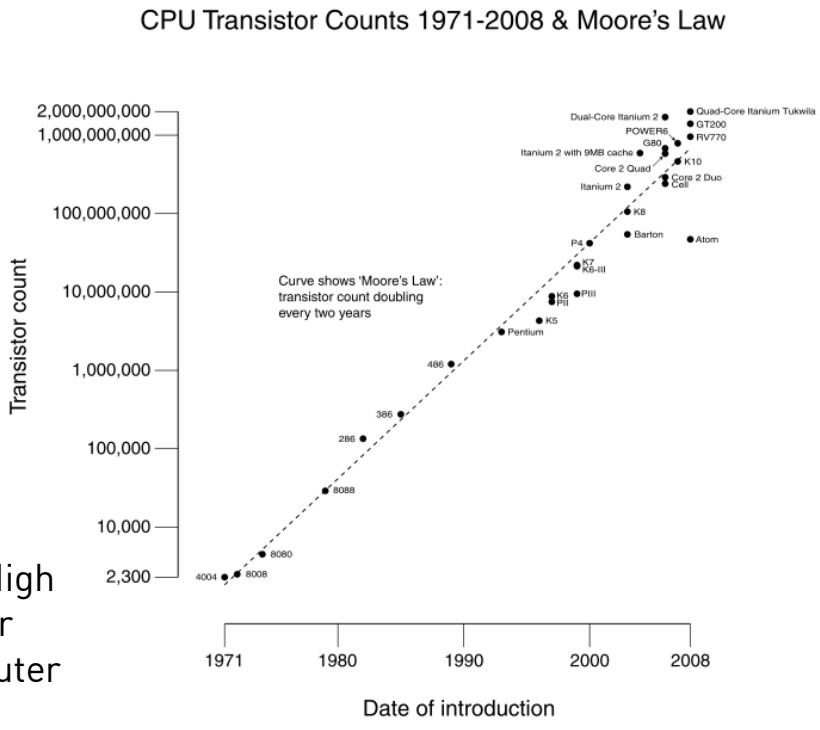
(de)Motivation: Moore's Law

Four Factors:

1) **Moore's Law**
2) **Amdahl's Law**
3) **Inherent Difficulty**
4) **Lack of a Unifying Paradigm**

The number of transistors per chip doubles every two years.

Result: Until recently, last year's High Performance Computer was slower than next year's Commodity Computer (exaggeration to make a point).



Source: "Moore's law." Wikipedia, The Free Encyclopedia. 15 Oct 2009 <http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=319734055>.

(de)Motivation: Amdahl's Law (1)

Four Factors:

1) Moore's Law
2) **Amdahl's Law**
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

Two Critical Measurements:

If $T(p)$ = total run-time with p processors

Then

Speedup = $S(p) = T(1) / T(p)$.

Efficiency = $E(p) = S(p) / p$.

By the way... Speedup and Efficiency are *really crucial* concepts! Remember them.

(de)Motivation: Amdahl's Law (2)

Four Factors:

1) Moore's Law
2) **Amdahl's Law**
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

Definitions:

$T(p)$ = total run-time with p processors

Speedup = $S(p) = T(1) / T(p)$.

Efficiency = $E(p) = S(p) / p$.

More Definitions:

α = inherently sequential time (e.g. input, output...)

π = parallelizable time.

Thus $T(1) = \alpha + \pi$

$T(p) = \alpha + \pi / p$

Let $f = \alpha / (\alpha + \pi)$ e.g. $1/4^{th}$

Then Speedup is...

$$S(p) = T(1) / T(p) = \frac{\alpha+\pi}{\alpha+\pi/p}$$

$$\ldots = \frac{1}{\left(\frac{\alpha+\pi/p}{\alpha+\pi}\right)} = \frac{1}{\frac{\alpha}{\alpha+\pi}+\frac{\pi/p}{\alpha+\pi}}$$

$$\ldots = \frac{1}{f+\frac{\pi}{p(\alpha+\pi)}} = \frac{1}{f+\frac{(1-f)}{p}}$$

Source: H. Bauke, S. Mertens, *Cluster Comuting,* Springer Verlag, 2006. Section 1.5 (p. 10-13).

(de)Motivation: Amdahl's Law (3)

Four Factors:

1) Moore's Law
2) **Amdahl's Law**
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

Definitions:

$T(p)$ = total run-time with p processors

Speedup = $S(p) = T(1) / T(p)$.

Efficiency = $E(p) = S(p) / p$.

More Definitions:

α = inherently sequential time (e.g. input, output...)

π = parallelizable time.

Thus $T(1) = \alpha + \pi$

$T(p) = \alpha + \pi / p$

Let $f = \alpha / (\alpha + \pi)$ e.g. $1/4^{th}$

Then Speedup is...

$$S(p) = T(1) / T(p) = \frac{\alpha+\pi}{\alpha+\pi/p}$$

The upper limit on Speedup is

$$S(p) = \frac{1}{f + \frac{(1-f)}{p}} \leqslant \frac{1}{f}$$

Source: H. Bauke, S. Mertens, *Cluster Computing,* Springer Verlag, 2006. Section 1.5 (p. 10-13).

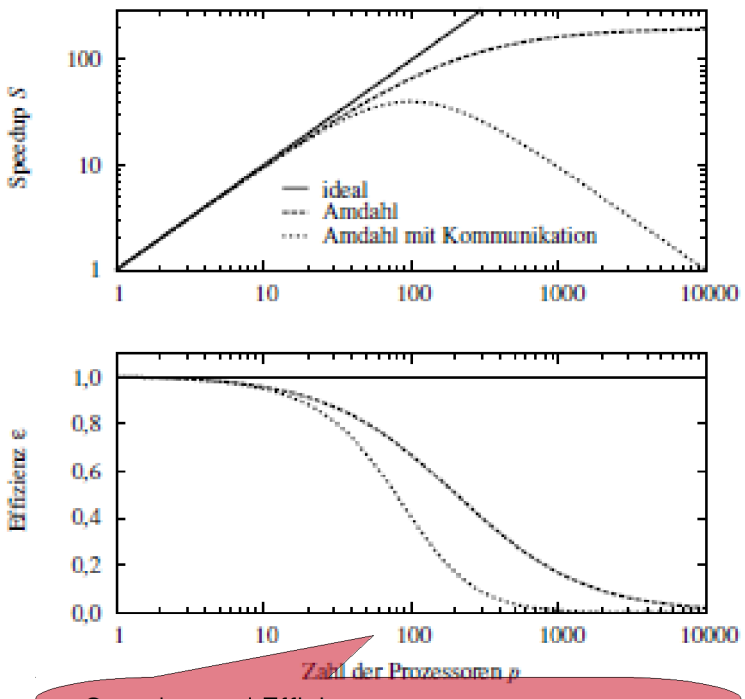
(de)Motivation: Amdahl's Law (4)

Four Factors:

1) Moore's Law
2) **Amdahl's Law**
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

Since the upper limit on Speedup is $1/f$, Efficiency goes to zero as $p \rightarrow \infty$

If we introduce *communication* into our model, we can find a value of p with maximum speedup. Afterwards, more processors mean more run-time!



Speedup and Efficiency, with and without communication costs, for $f = 0.005$, and $1 \leq p \leq 10000$

Source: H. Bauke, S. Mertens, *Cluster Computing,* Springer Verlag, 2006. Abb. 1.3, p. 12.

(de)Motivation: Inherent Difficulty (1)

Four Factors:

1) Moore's Law
2) Amdahl's Law
3) **Inherent Difficulty**
4) Lack of a Unifying Paradigm

Prima Facie Argument:

- Sequential Programmers have 2 things to worry about:
  1) Space (Memory)
  2) Time (Instructions)
- Parallel Programmers have 3 things to worry about:
  1) Space (Memory)
  2) Time (Instructions)
  3) Choice of Processor (Distributing the instructions – and possibly the memory – across the processors).

(de)Motivation: Inherent Difficulty (2)

Four Factors:

1) Moore's Law
2) Amdahl's Law
3) **Inherent Difficulty**
4) Lack of a Unifying Paradigm

An Example: **Sequential Scheduling**

Problem: Given a set of n jobs, each of which takes time t_i, assign an order to the jobs to minimize total run time.

Solution: Trivial. Any order will do. The run time is always the same:

$$\sum_{0 \leq i < n} t_i$$

Comparison: **Parallel Scheduling**

Problem: Given a set of n jobs, each of which takes time t_i, and m machines, assign each job to a machine so as to minimize total run time.

Solution: Finding an optimal schedule is ***NP-Complete!!!***

(de)Motivation: Lack of a Unifying Paradigm

Four Factors:

1) Moore's Law
2) Amdahl's Law
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

"Some Parallel Programming Environments from the Mid-1990s"

(How) Is this really different from the sequential languages?

"C* in C	CUMULVS	Java RMI	P-RIO	Quake
ABCPL	DAGGER	javaPG	P3L	Quark
ACE	DAPPLE	JAVAR	P4-Linda	Quick Threads
ACT++	Data Parallel C	JavaSpaces	Pablo	Sage++
ADDAP	DC++	JIDL	PADE	SAM
Adl	DCE++	Joyce	PADRE	SCANDAL
Adsmith	DDD	Karma	Panda	SCHEDULE
AFAPI	DICE	Khoros	Papers	SciTL
ALWAN	DIPC	KOAN/Fortran-S	Para++	SDDA
AM	Distributed Smalltalk	LAM	Paradigm	SHMEM
AMDC	DOLIB	Legion	Parafrase2	SIMPLE
Amoeba	DOME	Lilac	Paralation	Sina
AppLeS	DOSMOS	Linda	Parallaxis	SISAL
ARTS	DRL	LiPS	Parallel Haskell	SMI
Athapascan-0b	DSM-Threads	Locust	Parallel-C++	SONiC
Aurora	Ease	Lparx	ParC	Split-C
Automap	ECO	Lucid	ParLib++	SR
bb_threads	Eilean	Maisie	ParLin	Sthreads
Blaze	Emerald	Manifold	Parlog	Strand
BlockComm	EPL	Mentat	Parmacs	SUIF
BSP	Excalibur	Meta Chaos	Parti	SuperPascal
C*	Express	Midway	pC	Synergy
C**	Falcon	Millipede	pC++	TCGMSG
C4	Filaments	Mirage	PCN	Telegraphos
CarlOS	FLASH	Modula-2*	PCP:	The FORCE
Cashmere	FM	Modula-P	PCU	Threads.h++
CC++	Fork	MOSIX	PEACE	TRAPPER
Charlotte	Fortran-M	MpC	PENNY	TreadMarks
Charm	FX	MPC++	PET	UC
Charm++	GA	MPI	PETSc	uC++
Chu	GAMMA	Multipol	PH	UNITY
Cid	Glenda	Munin	Phosphorus	V
Cilk	GLU	Nano-Threads	POET	ViC*
CM-Fortran	GUARD	NESL	Polaris	Visifold V-NUS
Code	HAsL	NetClasses++	POOL-T	VPE
Concurrent ML	HORUS	Nexus	POOMA	Win32 threads
Converse	HPC	Nimrod	POSYBL	WinPar
COOL	HPC++	NOW	PRESTO	WWWinda
CORRELATE	HPF	Objective Linda	Prospero	XENOOPS
CparPar	IMPACT	Occam	Proteus	XPC
CPS	ISETL-Linda	Omega	PSDM	Zounds
CRL	ISIS	OOF90	PSI	ZPL
CSP	JADA	Orca	PVM	
Cthreads	JADE	P++	QPC++	

Source: T. Mattson, B. Sanders, B. Massingill, **Patterns of Parallel Programming,** Addison-Wesley, 2005. Table 2.1, p. 14.

Outline of Chapter I – Introduction

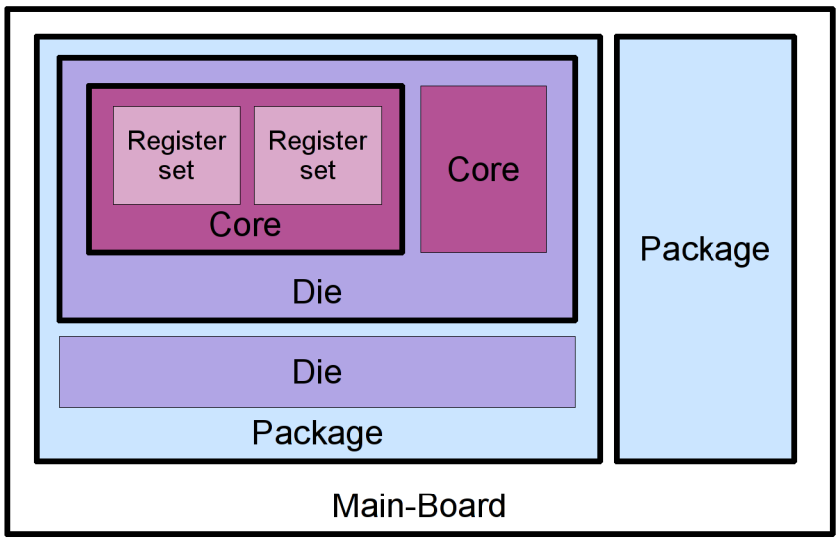
I Introduction
II Models of Parallel Computing
III Parallel Computation Design
IV The Message-Passing Paradigm
V The Shared Memory Paradigm
VI Frontiers
VII Summary

A) Personal Introductions
B) What is Parallel & Distributed Computing?
C) Demotivation: Why wasn't Parallelism used?
D) Motivation: Why use Parallelism?
E) Paradigms & Packages for Parallel Programming: An Overview & an Outlook
F) Summary & Bibliography

Motivation(!): Moore meets Multicores

Four Factors:

1) **Moore's Law**
2) **Amdahl's Law**
3) **Inherent Difficulty**
4) **Lack of a Unifying Paradigm**



The number of transistors keeps growing, but they can't all be used in one CPU any more:

- Deep Pipelining has its limits, this led to *hyper-threading* (virtually more than 1 CPU)
- Hyper-threading has its limits, this led to *multicore chips* (really more than 1 CPU per chip).
- Both can be combined.

Result: Sequential Processors are dying out! Multiprocessors are now the rule, not the exception!

Motivation(!): Amdahl meets Gustafson

Four Factors:

1) **Moore's Law**
2) **Amdahl's Law**
3) **Inherent Difficulty**
4) **Lack of a Unifying Paradigm**

By the way, Gustafson *published* it, but said it was due to E. Barsis

Gustafson's Law:

Speedup is...

$$S(p) = T(1) / T(p) = \frac{f\,T(p) + p(1-f)T(p)}{T(p)}$$

$$= f + p(1-f) = \boldsymbol{p + (1-p)f}$$

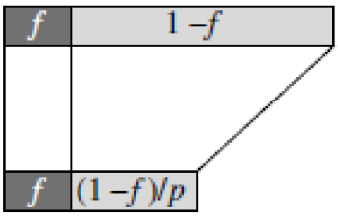
Amdahl:

Speedup is...

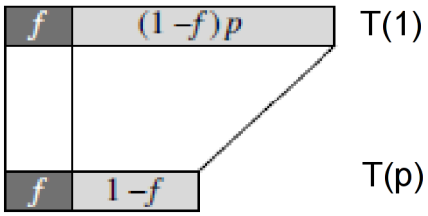
$$S(p) = T(1) / T(p)$$

$$= \frac{1}{f + \frac{(1-f)}{p}}$$

Amdahl



Gustafson



Moral of the Story:

- Speedup is limited only if T(1) is held constant;
- For sufficiently large T(1), f goes to zero

Source: H. Bauke, S. Mertens, *Cluster Comuting,* Springer Verlag, 2006. Section 1.7 and Abb. 1.6, p. 16.

Motivation(!): Isoefficiency

Four Factors:

1) Moore's Law
2) **Amdahl's Law**
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

The Isoefficiency Equation:

Constant Efficiency implies that

$$T(n,1)=\frac{E(n,p)}{1-E(n,p)}T_0(n,p)$$

$$T(n,1)=C\,T_0(n,p)$$

Can we extend Amdahl's Law to express how problems *scale*?

New definition of parallel run time:

$$T(n,p)=\alpha(n)+\frac{\pi(n)}{p}+T_c(n,p)$$

$T_c(n,p)$ is the communication costs + overhead.

It follows that efficiency is

$$E(n,p)=T(n,1)/pT(n,p)$$

$$= \frac{\alpha(n)+\pi(n)}{p\alpha(n)+\pi(n)+pT_c(n,p)}$$

$$= \frac{\alpha(n)+\pi(n)}{\alpha(n)+\pi(n)+T_0(n,p)}$$

Where $T_0(n,p)=(p-1)\alpha(n)+pT_c(n,p)$

Motivation(!): Amdahl meets Disney/Pixar

Four Factors:

1) Moore's Law
2) **Amdahl's Law**
3) Inherent Difficulty
4) Lack of a Unifying Paradigm

Moral of the Story:

- **Problems scale.**
- There are problems out there that will gladly take all the processing power we can give them.

1995: Disney/Pixar release *Toy Story*.
Length: 51 minutes, 44 Seconds, 24 Frames per second.
Rendered on a "render farm" with 100 dual-processor workstations.

1999: Disney/Pixar release *Toy Story 2*.
Length: 92 Minutes.
Rendered on a 1,400-processor system.

2001: Disney/Pixar release *Monsters, Inc.*
Length: 94 Minutes.
Rendered on a system with 250 enterprise servers, each with 14 processors for a total of 3,500 processors.

Source: T. Mattson, B. Sanders, B. Massingill, **Patterns of Parallel Programming,** Addison-Wesley, 2005, p. 1-2.

Motivation(!): Embarrassing Parallelism

Four Factors:

1) Moore's Law
2) Amdahl's Law
3) **Inherent Difficulty**
4) Lack of a Unifying Paradigm

Clay Breshears ("The Art of Concurrency", O Reilly, 2009) suggests saying "enchantingly parallel" instead of "embarssingly parallel".

Of course, not all problems are embarrassingly/enchantingly parallel, and we don't want to restrict ourselves to those that are. The others are fun too!

Example: Parallel Scheduling is NP-Complete...

Counter-Example 1: Finding *the* optimal schedule is very hard; finding *a nearly optimal* schedule is easy. Longest-Processing-Time first scheduling is a "4/3-optimal solution" (never worse than 4/3 the optimum).

Counter-Example 2: Given sufficiently many, sufficiently small jobs, any schedule approximates the optimum.

Definition: An *embarrassingly parallel* problem is a problem consisting of very many (essentially) independent tasks. Like rendering Pixar films!

Motivation(!): Something like Convergence

Four Factors:

1) Moore's Law
2) Amdahl's Law
3) Inherent Difficulty
4) **Lack of a Unifying Paradigm**

Of all of these, only **Java...**

...and **MPI** have survived.

plus Newcomers: **C++11 (Boost)**, **OpenMP** & *<something for GPGPUs>*

...more or less...

"C* in C	CUMULVS	Java RMI	P-RIO	Quake
ABCPL	DAGGER	javaPG	P3L	Quark
ACE	DAPPLE	JAVAR	P4-Linda	Quick Threads
ACT++	Data Parallel C	JavaSpaces	Pablo	Sage++
ADDAP	DC++	JIDL	PADE	SAM
Adl	DCE++	Joyce	PADRE	SCANDAL
Adsmith	DDD	Karma	Panda	SCHEDULE
AFAPI	DICE	Khoros	Papers	SciTL
ALWAN	DIPC	KOAN/Fortran-S	Para++	SDDA
AM	Distributed Smalltalk	LAM	Paradigm	SHMEM
AMDC	DOLIB	Legion	Parafrase2	SIMPLE
Amoeba	DOME	Lilac	Paralation	Sina
AppLeS	DOSMOS	Linda	Parallaxis	SISAL
ARTS	DRL	LiPS	Parallel Haskell	SMI
Athapascan-0b	DSM-Threads	Locust	Parallel-C++	SONiC
Aurora	Ease	Lparx	ParC	Split-C
Automap	ECO	Lucid	ParLib++	SR
bb_threads	Eilean	Maisie	ParLin	Sthreads
Blanc	Emerald	Manifold	Parlog	Strand
BlockComm	EPL	Mentat	Parmacs	SUIF
BSP	Excalibur	Meta Chaos	Parti	SuperPascal
C*	Express	Midway	pC	Synergy
C**	Falcon	Millipede	pC++	TCGMSG
C4	Filaments	Mirage	PCN	Telegraphos
CarlOS	FLASH	Modula-2*	PCP:	The FORCE
Cashmere	FM	Modula-P	PCU	Threads.h++
CC++	Fork	MOSIX	PEACE	TRAPPER
Charlotte	Fortran-M	MpC	PENNY	TreadMarks
Charm	FX	MPC++	PET	UC
Charm++	GA	MPI	PETSc	uC++
Chu	GAMMA	Multipol	PH	UNITY
Cid	Glenda	Munin	Phosphorus	V
Cilk	GLU	Nano-Threads	POET	ViC*
CM-Fortran	GUARD	NESL	Polaris	Visifold V-NUS
Code	HAsL	NetClasses++	POOL-T	VPE
Concurrent ML	HORUS	Nexus	POOMA	Win32 threads
Converse	HPC	Nimrod	POSYBL	WinPar
COOL	HPC++	NOW	PRESTO	WWWinda
CORRELATE	HPF	Objective Linda	Prospero	XENOOPS
CparPar	IMPACT	Occam	Proteus	XPC
CPS	ISETL-Linda	Omega	PSDM	Zounds
CRL	ISIS	OOF90	PSI	ZPL
CSP	JADA	Orca	PVM	
Cthreads	JADE	P++	QPC++	

Outline of Chapter I – Introduction

I Introduction

II Models of Parallel Computing

III Parallel Computation Design

IV The Message-Passing Paradigm

V The Shared Memory Paradigm

VI Frontiers

VII Summary

A) Personal Introductions

B) What is Parallel & Distributed Computing?

C) Demotivation: Why wasn't Parallelism used?

D) Motivation: Why use Parallelism?

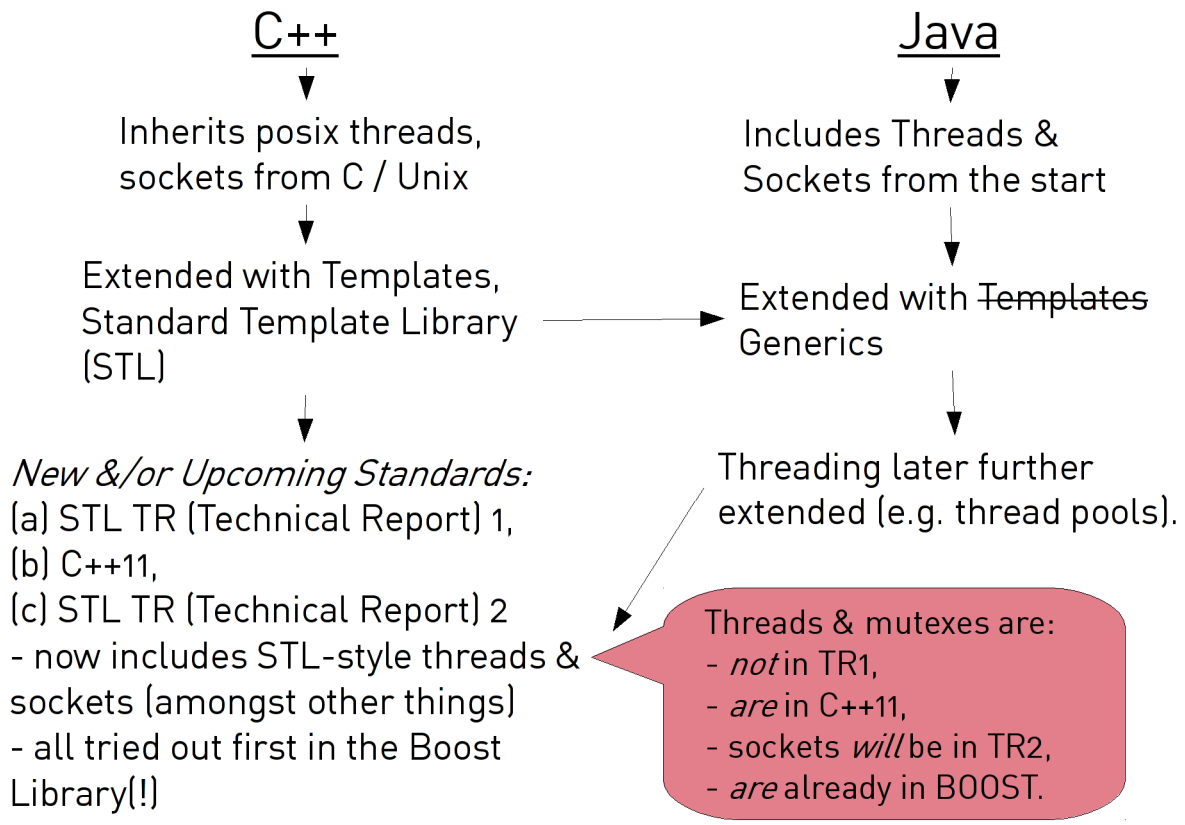
E) Paradigms & Packages for Parallel Programming: An Overview & an Outlook

F) Summary & Bibliography

Paradigms vs. Programming Packages

Paradigm \ Package	Java / C++0x (Boost)	OpenMP	MPI
Shared Memory	Yes	Yes	No (*)
Message Passing	Yes	No	Yes
Programming Language Extension	No (Java) No* (C++0x)	Debatably Yes (Compiler-Extensions)	No (!)
Libraries	Yes	Yes	Yes
Languages:	Java / C++	C, (C++), Fortran	C, (C++), Fortran, Java, ...

Threads in C++11: History



Threads in C++11 (1)

C++11 supports:

1) Threads
2) Synchronization
3) Thread-local Memory
4) (Sockets for Message Passing)

```
#include <thread>
void threadFunction() {
    std::cout << "Hello from Thread "
              << std::this_thread::get_id()
              << std::endl;
}

int main()
{
  std::thread t1(threadFunction);
  std::thread t2(threadFunction);
  t1.join();
  t2.join();
}
```

Function pointer(!)

Std::thread Constructor starts threads running, join method waits until a thread is finished.

We have a synchronization problem here. Where?

Source: http://en.highscore.de/cpp/boost/multithreading.html

Threads in C++ (2)

Boost supports:

1) Threads
2) Synchronization
3) Thread-local Memory
4) (Sockets for Message Passing)

```
#include <thread>
// global variable, shared by all threads
std::mutex globalMutex;
void threadFunction() {
    globalMutex.lock(); // critical section
    std::cout << "Hello from Thread "
              << std::this_thread::get_id()
              << std::endl;
    globalMutex.unlock(); // parallelism resumes
}
int main()
{
  std::thread t1(threadFunction);
  std::thread t2(threadFunction);
  t1.join();
  t2.join();
}
```

Mutex = Mutual Exclusive. Mutexes are used to guard *critical sections* – areas with limited (or no) parallelism.

Source: http://en.highscore.de/cpp/boost/multithreading.html

Threads in C++11 (3)

Boost supports:

1) Threads
2) Synchronization
3) Thread-local Memory
4) Sockets for Message Passing

```
#include <thread>

void init_number_generator() {
  thread_local static bool doneOnce(false);
  If (! doneOnce )   {
    doneOnce = true;
    std::srand(static_cast<unsigned
   int>(std::time(0)));
  }
} // end number_generator

std::mutex globalMutex;

void random_number_generator()  {
  init_number_generator();
  int i = std::rand();
  std::lock_guard<std::mutex> lock(mutex);
  std::cout << i << std::endl;
} // end random_number_generator


int main() {
  std::thread t[3];
  for (int i = 0; i < 3; ++i)
    t[i] = std::thread(random_number_generator);
  for (int i = 0; i < 3; ++i)
    t[i].join();
}
```

"thread_local" is a new keyword in C++11!

Warning: Compiler support still largely missing!
This code is untested!

We need more than static (global) and local variables!

lock_guard: Constructor locks, destructor unlocks – Principle: RAII = Resource Acquisition Is Initialization

Footnote: C++11 has new random number generators too...

Source: http://en.highscore.de/cpp/boost/multithreading.html

Boost Threads in C++ (4)

Boost supports:

1) Threads
2) Synchronization
3) Thread-local Memory
4) Sockets for Message Passing

Sockets are part of the more general "asynchronous I/O" (asio) library.

Central concepts: IO_Services & Handlers

This example connects to a web server (via tcp) and downloads an html page.

```
#include <boost/asio.hpp>
#include <boost/array.hpp>
boost::asio::io_service io_service;
boost::asio::ip::tcp::resolver resolver(io_service);
boost::asio::ip::tcp::socket sock(io_service);
boost::array<char, 4096> buffer;
void read_handler(const boost::system::error_code &ec,
   std::size_t bytes_transferred) {
  if (!ec)   {
    std::cout << std::string(buffer.data(), bytes_transferred)
   << std::endl;
    sock.async_read_some(boost::asio::buffer(buffer),
   read_handler);
  }
} // end read_handler
void connect_handler(const boost::system::error_code &ec) {
  if (!ec)   {
    boost::asio::write(sock, boost::asio::buffer("GET / HTTP
   1.1\r\nHost: highscore.de\r\n\r\n"));
    sock.async_read_some(boost::asio::buffer(buffer),
   read_handler);
  }
} // end connect_handler
void resolve_handler(const boost::system::error_code &ec,
                     boost::asio::ip::tcp::resolver::iterator
   it) {
  if (!ec) {
    sock.async_connect(*it, connect_handler);
  }
} // end resolve_handler
int main() {
  boost::asio::ip::tcp::resolver::query
   query("www.highscore.de", "80");
  resolver.async_resolve(query, resolve_handler);
  io_service.run();
}
```

Source: http://en.highscore.de/cpp/boost/asio.html

OpenMP: Compiler Supported Parallelism (1)

OpenMP supports

1) Threads
2) Synchronization with private and shard data, fork/join
3) Loop Parallelism

OpenMP is a compiler-extension (not really a language extension) for C, C++ and Fortran. Assumes shared memory (usually).

Supported by the Intel and gnu compilers (et. al.).

Not to be confused with OpenMPI!

```
#include <omp.h>
#include <iostream>

main ()
{
  // serial startup goes here...
  #pragma omp parallel num_threads (8)
  {    // parallel segment
    printf(
      "\nHello world, I am thread %d\n",
      omp_get_thread_num() );
  }
 // rest of serial segment ...
}
```

OpenMP: Compiler Supported Parallelism (2)

OpenMP supports

1) Threads
2) Synchronization with private and shard data, fork/join
3) Loop Parallelism

OpenMP encourages a *Single Program, Multiple Data* style, where *parallel blocks* follow each other sequentially (first *fork*, then *join*...).

```
#include <omp.h>
#include <iostream>

int a, b, sum;

main () {
  // serial segment
  b = 1;   a = 1;   sum = 0;

#pragma omp parallel num_threads (8) private (a) \
                    shared (b) reduction(+:sum)
  {     // parallel segment
  a += 1;   b += 1;     sum = 1;
  printf("\nThread%d  a=%d, b=%d, sum=%d",
         omp_get_thread_num(), a, b, sum);
  }

  // rest of serial segment
  printf("  Thread%d  a=%d, b=%d, sum=%d\n",
         omp_get_thread_num(), a, b, sum);

#pragma omp parallel num_threads (8) private (a) \
                    shared (b) reduction(+:sum)
  {     // parallel segment
  a += 1;     b += 1;     sum = 1;
  printf("\nThread%d  a=%d, b=%d, sum=%d",
          omp_get_thread_num(), a, b, sum);
  }

  // rest of serial segment
  printf("  Thread%d  a=%d, b=%d, sum=%d\n",
         omp_get_thread_num(), a, b, sum);

}
```

Thread0 a=1, b=9, sum=8

Thread0 a=1, b=17, sum=16

OpenMP: Compiler Supported Parallelism (3)

OpenMP supports

1) Threads
2) Synchronization with private and shard data, fork/join
3) Loop Parallelism

OpenMP provides semi-automatic parallelization of loops. This is the dominant programming style in OpenMP.

```
#include <omp.h>
#include <iostream>

int sum;
float a[100], b[100];

main ()
{
  // serial segment
  sum = 0;

  // Initialise array a
  for (int j = 0; j < 100; ++j) a[j] = j;

  #pragma omp parallel  num_threads (2)
  {    // parallel segment

    #pragma omp for
    for (int i = 0; i < 100; ++i)
        b[i] = a[i] * 2.0;

  }

  // rest of serial segment
  // Print array a
  for (int k = 0; k < 100; ++k)
     printf("%5.2f ", b[k]);
}
```

MPI: Compiler Supported Parallelism (1)

MPI supports

1) Remote Execution (not threads)
2) Message Passing
3) Data Parallelism

MPI is a **library standard**, defined for many languages (including Fortran, C and C++ - shown here), available in various implementations (e.g. Open MPI, not to be confused with OpenMP).

It does not assume shared memory – but uses it where available.

Every MPI Program should start with MPI::Init() and end with MPI::Finalize().

```
#include <mpi.h>
int main(int argc, char **argv)
{
    int rank, size;
    MPI_Init();
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    printf(
     "Hello, world!  I am %d of %d.\n",
      rank, size );

    MPI_Finalize();
    return 0;
}
```

The number of parallel programs is determined by how the programs are run – e.g. with the command **mpiexec**.

MPI: Compiler Supported Parallelism (2)

MPI supports

1) Remote Execution (not threads)
2) Message Passing
3) Data Parallelism

The basic operations in MPI are **sending and receiving messages.**

Message passing can be blocking or non-blocking. Functions are available for handling arbitrary data – and ensuring compatibility across heterogeneous machines.

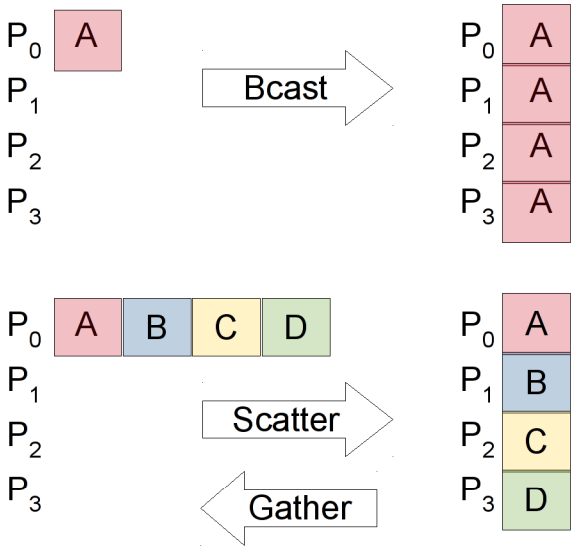
```
#include <mpi.h>
int main(int argc, char *argv[]) {
    int rank, size, next, prev, message, tag = 201;
    MPI_Status stat;
    MPI_Init();
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    next = (rank + 1) % size;
    prev = (rank + size - 1) % size;
    message = 10;
    if (0 == rank)
       MPI_Send(&message, 1, MPI_INT, next, tag, MPI_COMM_WORLD);
    while (1) {
        MPI_Recv(&message, 1, MPI_INT, prev, tag, MPI_COMM_WORLD, &stat);
        if (0 == rank) –message;
        MPI_Send(&message, 1, MPI_INT, next, tag);
        if (0 == message) break; // exit loop!
    } // end while
    if (0 == rank)
        MPI_Recv(&message, 1, MPI_INT, prev, tag, MPI_COMM_WORLD, &stat);
    MPI_Finalize();
    return 0;
}
```

MPI: Compiler Supported Parallelism (3a)

MPI supports

1) Remote Execution (not threads)
2) Message Passing
3) Data Parallelism

MPI also contains message passing functions that make it easy to distribute data (usually arrays) across a group of processors.

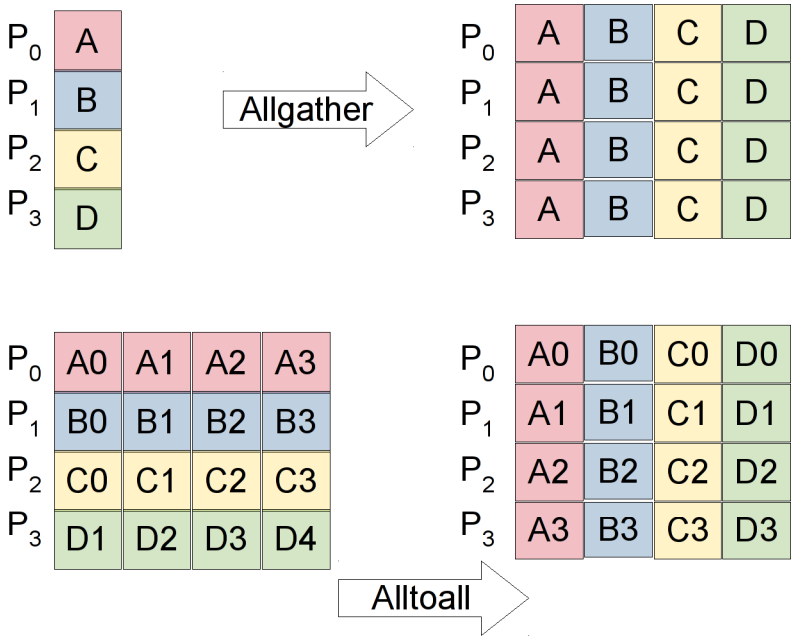


MPI: Compiler Supported Parallelism (3b)

MPI supports

1) Remote Execution (not threads)
2) Message Passing
3) Data Parallelism

There are more, this is not a complete list.



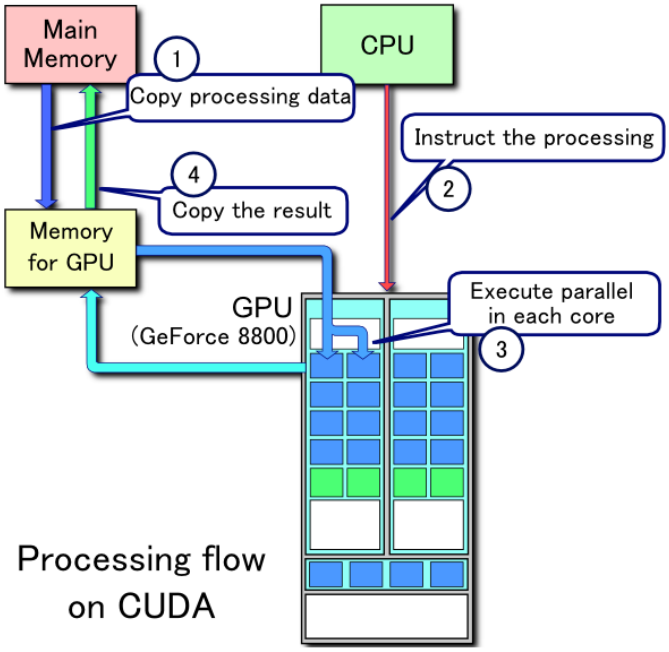
Outlook:

The cutting edge?
Specialized Hardware:

- General Purpose Graphics Processing Units (GPGPUs)
- Cell Processors
- Field Programmable Gate Arrays (FPGAs)

Problem: How to program?

- CUDA? Looks promising, but only NVIDIA
- OpenCL? Looks promising, promises to support many architectures...
- ...?!?



Processing flow on CUDA

Source: "CUDA." Wikipedia, The Free Encyclopedia. 28 Oct 2009, <http://en.wikipedia.org/w/index.php?title=CUDA&oldid=322466787>.

Outline of Chapter I – Introduction

Summary (of Chap. 1)

A) Personal Introductions

B) What is Parallel & Distributed Computing?

- Distributed Computing is a subset of Parallel Computing.
- High Performance Computing (HPC) is another subset.
- Parallel is strictly speaking a subset of concurrent.

C) (De)Motivation: Why (not) Use Parallelism?

- Moore's Law – vs. Multicore Processors
- Amdahl's Law – vs. Gustafson's Law
- Inherent Difficulty – vs. Embarassing (enchanting) Parallelism
- Lack of a Unifying Paradigm – vs. Convergence to...

Key Measurements: **Speedup** & **Efficiency**

D) Paradigms & Packages for Parallel Programming: An Overview & an Outlook

- *C++11 (& Boost)* offers threads & synchronization (mutexes etc.) for shared memory programming and sockets for message passing (C++).
- *OpenMP* offers shared memory programming with Loop Parallelism for languages with extended compilers (C, C++ & Fortan)
- *MPI* offers message passing programming with data parallelism in the form of a library for multiple languages (mainly C, C++ & Fortan).

Bibliography

Main Sources:

- H. Bauke, S. Mertens, **Cluster Computing,** Springer Verlag, 2006.
- Clay Breshears, **The Art of Concurrency,** O Reilly Media Inc, 2009.
- Ian Foster, **Designing and Building Parallel Programs,** Addison-Wesley Publishing, 1995.
  Available on-line: http://www.mcs.anl.gov/~itf/dbpp/(!)
- A. Grama, A. Gupta, G. Karypis, V. Kumar**, Introduction to Parallel Computing,** 2nd Edition, Addison Wesley (Pearson), 2003.
- T. G. Mattson, B. A. Sanders & B. L. Massingill, **Patterns for Prallel Programming,** Addison-Wesley (Pearson Education), 2005
- R. C. Moore, **SDAARC A Self Distributing Associative Architecture,** Shaker Verlag, 2001 – *Hard to find!* ;-)
- *This list is not meant to be exhaustive.*